



# Integração SQL – PL/SQL

A linguagem SQL (*Structured Query Language*) é um padrão em bancos de dados relacionais. Ela é uma **linguagem de domínio específico** (DSL – *Domains Specific Language*), isto é, ela foi criada com o propósito específico de manipular objetos em bancos de dados relacionais. Também é uma **linguagem declarativa**. Isto significa que os "programas" (*scripts*) em SQL se preocupam em definir o "que" e não o "como".

A linguagem SQL é dividida em sublinguagens, de acordo com o tipo de operação executada. A divisão mais granular define 5 sublinguagens:

- **DDL** (***D**ata **D**efinition **L**anguage*) – Utilizada na manipulação (criação, alteração, exclusão) de **objetos** do banco de dados (comandos CREATE, ALTER e DROP);
- **DML** (***D**ata **M**anipulation **L**anguage*) – Utilizada para manipulação do **conteúdo** de objetos do banco de dados, ou seja, os dados propriamente ditos (comandos INSERT, UPDATE e DELETE);
- **DQL** (***D**ata **Q**uery **L**anguage*) – Utilizada na **consulta** dados (comando SELECT);
- **TCL** (***T**ransaction **C**ontrol **L**anguage*) – Utilizada no controle de **transações** (comandos SET TRANSACTION

START TRANSACTION, COMMIT, ROLLBACK e SAVEPOINT);

- **DCL** (***D****ata* ***C****ontrol* ***L****anguage*) – Utilizada no controle da segurança dos dados, atribuindo permissões e privilégios de usuários (comandos GRANT, REVOKE e DENY).

Pelo fato de ter sido desenvolvida especificamente para "rodar" em um SGBDR, a linguagem PL/SQL se integra de forma muito natural com SQL. No entanto, algumas restrições se aplicam. Comandos DDL, utilizados na criação, alteração e exclusão de objetos, não são permitidos em programas PL/SQL.[1] Já os comandos DML e DQL, os mais utilizados no desenvolvimento de aplicações, são permitidos.

## INCLUSÃO DE REGISTROS

O comando INSERT é utilizado para a inserção de novos registros em uma tabela. Sua forma geral é:

```
INSERT [INTO] nome_da_tabela
    [(coluna1 [, coluna2] ...)]
    {VALUES | VALUE} (valor1, [, valor2]) [, (valor1, [, valor2])] ...
```

A relação (*coluna1*, *coluna2*, ...) é opcional. Quando ela é utilizada, não é necessário colocar os nomes de todas as colunas, mas apenas daquelas que receberão valores (as demais recebem os valores padrão ou NULL). Agora, quando não utilizada, devem ser fornecidos valores para todas as colunas, na ordem em que estão na tabela. O exemplo a seguir ilustra a utilização do comando INSERT em uma *procedure*. (os *scripts* SQL para a criação das tabelas encontram-se no material de apoio).



```
CREATE OR REPLACE PACKAGE pkg_filmes AS
    FILME_REPETIDO EXCEPTION;
    CAMPO_NULO EXCEPTION;
    PRAGMA EXCEPTION_INIT(FILME_REPETIDO, -1);
    PRAGMA EXCEPTION_INIT(CAMPO_NULO, -2290);
END pkg_filmes;

CREATE OR REPLACE FUNCTION inclui_filme (
    filme_id NUMBER,
    titulo   VARCHAR2,
    diretor  VARCHAR2,
    ano      NUMBER,
    pais     VARCHAR2,
    duracao  NUMBER
)
RETURN NUMBER
IS
    ret NUMBER := 0;
-- Inclui novo registro e trata as exceções que podem ocorrer. Retorna 0
-- (inclusão ok) ou o código do erro (inclusão nok).
BEGIN
    -- Para poder retornar ao corpo da função, após o tratamento do erro,
    -- coloca-se o comando INSERT dentro de um bloco.
    BEGIN
        INSERT INTO filmes VALUES (filme_id, titulo, diretor, ano, pais, duracao);
    EXCEPTION
        WHEN pkg_filmes.FILME_REPETIDO THEN
            ret := -1;
        -- Um campo NOT NULL recebe o calor NULL
        WHEN pkg_filmes.CAMPO_NULO THEN
            ret := -2290;
        WHEN OTHERS THEN
            ret := SQLCODE;
    END;
    RETURN ret;
END;

DECLARE
    ret NUMBER;
BEGIN
    ret := inclui_filme(1, 'Caçadores da Arca Perdida','Steven Spielberg',1981,'E.U.A.',115);
    IF ret = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Inclusão bem sucedida.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Erro na inclusão: ' || ret);
    END IF;
END;
```

A mensagem 'Inclusão bem sucedida.' é exibida após a execução do bloco anônimo.

É possível a utilização do tipo RECORD no lugar dos campos individuais da tabela *FILMES* tanto na passagem de parâmetros quanto no comando INSERT. A *procedure inclui_filme* será colocada dentro da *package pkg_filmes*. Além disto, o tratamento de exceções foi alterado a fim de retornar ao programa chamador o código e a mensagem de erro. A seguir, a versão modificada do exemplo anterior.

```
CREATE OR REPLACE PACKAGE pkg_filmes AS
  TYPE tipo_reg_filmes IS RECORD (
      filme_id filmes.filme_id%TYPE,
      titulo filmes.titulo%TYPE,
      diretor filmes.diretor%TYPE,
      ano filmes.ano%TYPE,
      pais filmes.pais%TYPE,
      duracao filmes.duracao%TYPE
  );
  CAMPO_NULO EXCEPTION;
  PRAGMA EXCEPTION_INIT(CAMPO_NULO, -2290);
  -- Inclui filme. Em caso de erro, retorna
  -- o código e mensagem de erro.
  PROCEDURE inclui_filme (
      registro IN pkg_filmes.TIPO_REG_FILMES,
      cod_erro OUT NUMBER,
      msg_erro OUT VARCHAR2
  );
END pkg_filmes;
```

```
CREATE OR REPLACE PACKAGE BODY pkg_filmes
AS
  PROCEDURE inclui_filme (
     registro IN pkg_filmes.TIPO_REG_FILMES,
     cod_erro OUT NUMBER,
     msg_erro OUT VARCHAR2
  )
  IS
  BEGIN
    BEGIN
      cod_erro := 0;
      INSERT INTO filmes VALUES registro;
    EXCEPTION
      WHEN DUP_VAL_ON_INDEX THEN
        cod_erro := SQLCODE;
        msg_erro := 'ID_FILME ou TITULO repetido';
      WHEN CAMPO_NULO THEN
        cod_erro := SQLCODE;
        msg_erro := 'TITULO ou DIRETOR nulo';
      WHEN OTHERS THEN
        cod_erro := SQLCODE;
        msg_erro := SQLERRM;
    END;
  END;
END pkg_filmes;
```

Observe que, na *package*, os tipos dos campos do RECORD foram definidos de forma diferente. Esta forma é chamada **ancoragem de tipo** e é mais uma boa característica da linguagem PL/SQL, dentro da filosofia orientada a modularidade e de esconder do desenvolvedor detalhes desnecessários. Como cada campo do RECORD corresponde a um campo da tabela, nada mais lógico do que atribuir ao primeiro o mesmo tipo do segundo. A forma geral é *tabela.campo%*TYPE. Recomenda-se que a ancoragem de tipos sempre que possível.

O bloco anônimo e o resultado exibido após sua execução são mostrados a seguir.



```
DECLARE
    registro pkg_filmes.TIPO_REG_FILMES;
    cod_erro NUMBER;
    msg_erro VARCHAR2(100);
BEGIN
    registro.filme_id := 4;
    registro.titulo   := 'Noites de Cabíria';
    registro.diretor  := 'Federico Fellini';
    registro.ano      := 1958;
    registro.pais     := 'Itália';
    registro.duracao  := 110;
    pkg_filmes.inclui_filme(registro, cod_erro, msg_erro);
    IF cod_erro = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Inclusão bem sucedida');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Erro na inclusão: ' || 'ORA' || TO_CHAR(cod_erro, '00000') ||
            ' (' || msg_erro || ')');
    END IF;
END;

Inclusão bem sucedida.
```

## ALTERAÇÃO DE REGISTROS

O comando UPDATE é utilizado para alteração de registros existentes em uma tabela. Sua forma geral é:

```
UPDATE nome_da_tabela
    SET coluna1 = [valor1 | expressão1 | DEFAULT]
        [,coluna2 = [valor2 | expressão2 | DEFAULT] ] ...
    [WHERE condição];
```

O valor atribuído pode ser uma constante, uma expressão ou o valor padrão para a coluna (DEFAULT). A cláusula SET atribui valores às colunas. Se a cláusula WHERE for incluída, apenas o(s) registro(s) em que *condição* for verdadeira serão afetados. Caso contrário, **todos** os registros da tabela serão afetados. O exemplo a seguir ilustra a utilização do comando UPDATE (apenas as alterações na *package* serão mostradas).

```
CREATE OR REPLACE PACKAGE pkg_filmes AS
    TYPE tipo_reg_filmes IS ...
    ...
    -- Altera filme por id_filme. Em caso de erro,
    -- retorna o código e a mensagem de erro.
    PROCEDURE altera_filme (
        registro filmes%ROWTYPE,
        cod_erro OUT NUMBER,
        msg_erro OUT VARCHAR2
    );
END pkg_filmes;
```

```
CREATE OR REPLACE PACKAGE BODY pkg_filmes
AS
  PROCEDURE inclui_filme (
  ...
  PROCEDURE altera_filme (
      registro filmes%ROWTYPE,
      cod_erro OUT NUMBER,
      msg_erro OUT VARCHAR2
  )
  IS
    x NUMBER;
  BEGIN
    BEGIN
      cod_erro := 0;
      SELECT COUNT(*) INTO x FROM filmes
      WHERE registro.filme_id = filme_id;
      IF x = 0 THEN
          RAISE NO_DATA_FOUND;
      END IF;
      UPDATE filmes
      SET titulo = registro.titulo,
          diretor = registro.diretor,
          ano = registro.ano,
          pais = registro.pais,
          duracao = registro.duracao
      WHERE filme_id = registro.filme_id;
    EXCEPTION
      WHEN DUP_VAL_ON_INDEX THEN
        cod_erro := SQLCODE;
        msg_erro := 'ID_FILME ou TITULO repetido';
      WHEN CAMPO_NULO THEN
        cod_erro := SQLCODE;
        msg_erro := 'TITULO ou DIRETOR nulo';
      WHEN OTHERS THEN
        cod_erro := SQLCODE;
        msg_erro := SQLERRM;
    END;
  END;
END pkg_filmes;
```

Observe duas coisas. Primeiro, foi utilizado o tipo *filmes%ROWTYPE* para o parâmetro *registro* no lugar de *TIPO_REG_FILMES.* A ancoragem de tipo, neste caso, atribuiu a *registro* um tipo RECORD com a mesma estrutura da tabela *filmes*. Esta é uma forma muito prática de ancoragem de tipos. Segundo, foi incluído o comando SELECT INTO antes do UPDATE. A função deste comando é simplesmente verificar se o registro a ser alterado existe em *FILMES*. O comando SELECT INTO atribui à variável *x* o valor retornado por COUNT(*). Se este valor for 0 (não existe o registro), a exceção NO_DATA_FOUND é gerada.

Uma forma mais compacta pode ser utilizada no comando UPDATE. No lugar de se atribuir valores às colunas uma a uma, o comando poderia ser escrito da forma a seguir.



```
UPDATE filmes SET ROW = registro WHERE filme_id = registro.filme_id;
```

O identificador ROW indica que todos os campos dos registros selecionados pela cláusula WHERE serão alterados, recebendo o conteúdo de *registro*.

O bloco anônimo e o resultado exibido após sua execução são mostrados a seguir.

```
DECLARE
    registro filmes%ROWTYPE;
    cod_erro NUMBER;
    msg_erro VARCHAR2(100);
BEGIN
    registro.filme_id := 4;
    registro.titulo   := 'Noites de Cabíria';
    registro.diretor  := 'Federico Fellini';
    registro.ano      := 1957; -- este campo foi alterado.
    registro.pais     := 'Itália';
    registro.duracao  := 110;
    pkg_filmes.altera_filme(registro, cod_erro, msg_erro);
    IF cod_erro = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Alteração bem sucedida');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Erro na alteração: ' || 'ORA' || TO_CHAR(cod_erro, '00000') ||
            ' (' || msg_erro || ')');
    END IF;
END;

Alteração bem sucedida.
```

## EXCLUSÃO DE REGISTROS

O comando DELETE é utilizado para remover registros. Muito cuidado: a cláusula WHERE **deve** ser utilizada para selecionar os registros a serem excluídos ou **todos** os registros da tabela serão excluídos. A sintaxe mais comum de DELETE é simples:

```
DELETE FROM nome_da_tabela
    [WHERE condição]
```



O exemplo a seguir implementa duas versões da função *exclui_registro*, utilizando *overloading* (apenas as alterações na *package* serão mostradas).

```
CREATE OR REPLACE PACKAGE pkg_filmes AS
    ...
    -- Exclui filme pelo id_filme.
    PROCEDURE exclui_filme (
        filme_id filmes.filme_id%TYPE,
        cod_erro OUT NUMBER,
        msg_erro OUT VARCHAR2
    );
    -- Exclui filme pelo titulo.
    PROCEDURE exclui_filme (
        titulo filmes.titulo%TYPE,
        cod_erro OUT NUMBER,
        msg_erro OUT VARCHAR2
    );END pkg_filmes;
```

```
CREATE OR REPLACE PACKAGE BODY pkg_filmes AS
    ...
    PROCEDURE exclui_filme (
        filme_id IN filmes.filme_id%TYPE,
        cod_erro OUT NUMBER,
        msg_erro OUT VARCHAR2
    )
    IS
      x NUMBER;
    BEGIN
      BEGIN
        cod_erro := 0;
        SELECT COUNT(*) INTO x FROM filmes
        WHERE filme_id = exclui_filme.filme_id;
        DELETE FROM filmes
        WHERE filme_id = exclui_filme.filme_id;
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
           cod_erro := SQLCODE;
           msg_erro := 'FILME_ID não existe';
         WHEN OTHERS THEN
           cod_erro := SQLCODE;
           msg_erro := SQLERRM;
      END;
    END;
  -----------------------------------------------------
```

```
    PROCEDURE exclui_filme (
        titulo filmes.titulo%TYPE,
        cod_erro OUT NUMBER,
        msg_erro OUT VARCHAR2
    )
    IS
      x NUMBER;
    BEGIN
      BEGIN
        cod_erro := 0;
        SELECT COUNT(*) INTO x FROM filmes
        WHERE titulo = exclui_filme.titulo;
        DELETE FROM filmes
        WHERE titulo = exclui_filme.titulo;
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
            cod_erro := SQLCODE;
            msg_erro := 'TITULO não existe';
         WHEN OTHERS THEN
            cod_erro := SQLCODE;
            msg_erro := SQLERRM;
      END;
    END;
END pkg_filmes;
```

O bloco anônimo e o resultado exibido após sua execução são mostrados a seguir.



```
DECLARE
    filme_id filmes.filme_id%TYPE;
    titulo filmes.titulo%TYPE;
    cod_erro NUMBER;
    msg_erro VARCHAR2(100);
BEGIN
    filme_id := 4;
    pkg_filmes.exclui_filme(filme_id, cod_erro, msg_erro);
    IF cod_erro = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Exclusão bem sucedida');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Erro na exclusão: ' || 'ORA' || TO_CHAR(cod_erro, '00000') ||
            ' (' || msg_erro || ')');
    END IF;
    titulo := 'Noites de Cabíria';
    pkg_filmes.exclui_filme(titulo, cod_erro, msg_erro);
    IF cod_erro = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Exclusão bem sucedida');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Erro na exclusão: ' || 'ORA' || TO_CHAR(cod_erro, '00000') ||
            ' (' || msg_erro || ')');
    END IF;
END;
```

```
Exclusão bem sucedida
Erro na exclusão: ORA 00100 (TITULO não existe)
```

## A CLÁUSULA *RETURNING*

A cláusula RETURNING INTO, quando usada em conjunto com os comandos INSERT, DELETE ou UPDATE permite que informações a respeito da execução do comando DML sejam atribuídos a variáveis. A sua forma geral é mostrada a seguir.

```
<comando DML> RETURNING expressão1 [, expressão 2] ... INTO var1 [, var 2] ...
```

*Expressão* pode envolver literais e/ou colunas da tabela referenciada no comando DML e a quantidade e o tipo das expressões deve ser o mesmo das variáveis. O exemplo a seguir cria a função *altera_pais_diretor*, que altera o país de um determinado diretor,

ambos passados como parâmetros, e retorna o número de registros afetados.



```
CREATE OR REPLACE PACKAGE pkg_filmes AS
    ...
    FUNCTION altera_pais_diretor (
        pais    filmes.pais%TYPE,
        diretor filmes.diretor%TYPE
    )
    RETURN NUMBER;
END pkg_filmes;
```

```
CREATE OR REPLACE PACKAGE BODY pkg_filmes AS
    ...
    FUNCTION altera_pais_diretor (
        pais    filmes.pais%TYPE,
        diretor filmes.diretor%TYPE
    )
    RETURN NUMBER
    IS
        registros_afetados NUMBER;
    BEGIN
        UPDATE filmes SET pais = altera_pais_diretor.pais
        WHERE diretor = altera_pais_diretor.diretor
        RETURNING COUNT(*) INTO registros_afetados;
        RETURN registros_afetados;
    END;
END pkg_filmes;
```

O bloco anônimo que chama esta função e o resultado exibido são mostrados a seguir.

```
-- Estes registros foram adicionados antes da execução do bloco anônimo
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (3, 'A Lista de Schindler',
    'Steven Spielberg',1993,'E.U.A.',195);
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (5, 'La Dolce Vitta',
    'Federico Fellini',1960,'França',114);
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (6, 'Amarcord',
    'Federico Fellini',1973,'França',123);
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (7, 'Terra em Transe',
    'Glauber Rocha',1967,'Portugal',111);
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (8, 'O Leão de 7 Cabecas',
    'Glauber Rocha',1970,'Portugal',103);
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (9, 'Jules e Jim',
    'François Truffaut',1962,'França',105);
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (10, 'Fahrenheit 451',
    'François Truffaut',1966,'França',112);
INSERT INTO filmes (FILME_ID, TITULO,DIRETOR,ANO,PAIS,DURACAO) VALUES (11, 'A Mulher do Lado',
    'François Truffaut',1981,'França',106);

DECLARE
    ret NUMBER;
BEGIN
    pkg_filmes.altera_pais_diretor('Federico Fellini', 'Itália');
    DBMS_OUTPUT.PUT_LINE('Registros alterados: ' || ret);
END;

Registros alterados: 2
```

Esta forma da cláusula RETURNING INTO deve ser utilizada quando se sabe que o comando afetará uma linha apenas ou quando são utilizadas funções de agregação para as linhas afetadas (SUM, AVG, MAX etc.). Se o comando associado afetar mais de uma linha da tabela e não forem utilizadas funções de agregação, será gerado o erro *ORA-01422: exact fetch returns more than requested number of rows*. Se nenhuma linha for afetada, as variáveis receberão valor NULL ou 0, no caso de SUM(*)

A cláusula RETURNING INTO também permite retornar resultados com mais de uma linha afetada. Nestes casos, deve ser usada a forma RETURNING BULK COLECT INTO. As variáveis que receberão os valores devem ser coleções. No exemplo a seguir, a função *altera_pais_diretor* é transformada em procedure e é modificada a fim de retornar também o título,  ano de lançamento e duração dos registros afetados.

```
CREATE OR REPLACE PACKAGE pkg_filmes AS
    ...
    PROCEDURE altera_pais_diretor (
      pais    IN filmes.pais%TYPE,
      diretor IN filmes.diretor%TYPE,
      r_titulo OUT TIPO_TAB_VARCHAR100,
      r_ano OUT TIPO_TAB_NUMBER,
      r_duracao OUT TIPO_TAB_NUMBER
    );
END pkg_filmes;
```

```
CREATE OR REPLACE PACKAGE BODY pkg_filmes AS
    ...
    PROCEDURE altera_pais_diretor (
        pais IN filmes.pais%TYPE,
        diretor IN filmes.diretor%TYPE,
        r_titulo OUT TIPO_TAB_VARCHAR100,
        r_ano OUT TIPO_TAB_NUMBER,
        r_duracao OUT TIPO_TAB_NUMBER

    )
    IS
    BEGIN
        UPDATE filmes SET pais = altera_pais_diretor.pais
        WHERE diretor = altera_pais_diretor.diretor
        RETURNING titulo, ano, duracao
        BULK COLLECT INTO r_titulo, r_ano, r_duracao;
    END;
END pkg_filmes;
```

Observe que não são necessários a inicialização e o uso do método EXTEND para as coleções. A própria cláusula RETURNING BULK COLLECT INTO se encarrega de inicializar e estender a coleção para o tamanho necessário.

Observe também que os tipos TIPO_TAB_VARCHAR100 e TIPO_TAB_NUMBER não foram declarados na *package*. Eles devem ser declarados globalmente ao esquema para que variáveis com estes tipos possam ser utilizadas em comandos SQL. Para declará-los no esquema, deve-se utilizar o comando CREATE TYPE, conforme mostrado a seguir.

```
CREATE OR REPLACE TYPE TIPO_TAB_VARCHAR100 IS TABLE OF VARCHAR2(100);
CREATE OR REPLACE TYPE TIPO_TAB_NUMBER IS TABLE OF NUMBER;
```

O bloco anônimo que chama a procedure e o resultado exibido após a execução são mostrados a seguir.

```
DECLARE
    titulo TIPO_TAB_VARCHAR100;
    ano TIPO_TAB_NUMBER;
    duracao TIPO_TAB_NUMBER;
    ind NUMBER;
BEGIN
    titulo := TIPO_TAB_VARCHAR100();
    ano := TIPO_TAB_NUMBER();
    duracao := TIPO_TAB_NUMBER();
    pkg_filmes.altera_pais_diretor('Brasil', 'Glauber Rocha', titulo, ano, duracao);
    DBMS_OUTPUT.PUT_LINE('Foram alterados ' || titulo.COUNT || ' registros:');
    DBMS_OUTPUT.NEW_LINE;
    FOR i IN 1 .. titulo.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Título:  '|| titulo(i));
        DBMS_OUTPUT.PUT_LINE('Ano:     '|| ano(i));
        DBMS_OUTPUT.PUT_LINE('Duração: '|| duracao(i));
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END;

Foram alterados 2 registros:

Título:  Terra em Transe
Ano:     1967
Duração: 111

Título:  O Leão de 7 Cabeças
Ano:     1970
Duração: 103
```

## TABELAS E TIPOS ESTRUTURADOS



Colunas de uma tabela podem ser definidas como tabelas aninhadas ou VARRAY. Desta forma, o SGBDR Oracle oferece suporte nativo a atributos multivalorados. A atribuição do tipo VARRAY a colunas de uma tabela é mais simples, já que este tipo tem limite em seu tamanho. Primeiro, deve-se criar um tipo através do comando CREATE TYPE. Tipos criados ficam armazenados e são globais ao esquema onde foram criados. Em seguida, atribui-se à coluna o tipo criado.

Tabelas aninhadas exigem algumas definições a mais, já que são não limitadas. Os exemplos a seguir ilustram a criação de tabelas com colunas de tipos estruturados.

```
CREATE OR REPLACE TYPE nome_dependente AS TABLE OF VARCHAR2(50);
CREATE OR REPLACE TYPE numero_telefone AS VARRAY(5) OF NUMBER;

CREATE TABLE cadastro (
    cadastro_id  NUMBER,
    nome VARCHAR2(50),
    telefones numero_telefone,
    dependentes nome_dependente
)
NESTED TABLE dependentes STORE AS dependentes_st;

INSERT INTO cadastro VALUES (1, 'Charles R. Darwin',
   numero_telefone(552122457864, 5521972345677),
   nome_dependente('William Erasmus Darwin', 'Anne Elizabeth Darwin', 'Mary Eleanor Darwin',
     'Henrietta Emma Darwin', 'George Howard Darwin', 'Elizabeth (Bessy) Darwin',
     'Francis Darwin', 'Leonard Darwin', 'Horace Darwin', 'Charles Waring Darwin'));

INSERT INTO cadastro VALUES (2, 'Albert Einstein',
   numero_telefone(552124690345, 5521965478723),
   nome_dependente('Eduard Einstein', 'Hans Albert Einstein'));

UPDATE cadastro SET dependentes = nome_dependente('Eduard Einstein', 'Hans Albert Einstein',
'Lieserl Einstein') WHERE cadastro_id = 2;

UPDATE cadastro SET telefones = numero_telefone(551122457864, 5511972345677) WHERE cadastro_id
= 1;
```

```
28  DECLARE
29      tel numero_telefone;
30      i NUMBER;
31  BEGIN
32      SELECT telefones INTO tel FROM cadastro WHERE cadastro_id = 1;
33      tel.EXTEND;
34      tel(tel.LAST) := 5511997662134;
35      UPDATE cadastro SET telefones = tel WHERE cadastro_id = 1;
36      i := tel.FIRST;
37      WHILE i IS NOT NULL LOOP
38          DBMS_OUTPUT.PUT_LINE(tel(i));
39          i := tel.NEXT(i);
40      END LOOP;
41  END;
```

Nas linhas 1 e 2 são declarados dois tipos globais. Nas linhas 5 a 10 a tabela cadastro é definida, com duas colunas (telefones e dependentes) de tipos estruturados. A linha 10 é necessária quando há colunas do tipo tabelas aninhada. Varrays são armazenados juntamente com as demais colunas da tabela, enquanto tabelas aninhadas são armazenadas separadamente. A cláusula NESTED TABLE STORE AS nomeia a estrutura de armazenamento da tabela aninhada. São necessárias tantas cláusulas quantas forem as tabelas aninhadas.

A inclusão de linhas na tabela é feita através do comando INSERT (linhas 12 a 20). Observe que os valores para as colunas telefones e dependentes devem ser inseridos utilizando-se seus respectivos *constructors*. A alteração de colunas segue a mesma lógica. Não é possível se alterar um elemento individual de colunas de tipos estruturados. Se isto for necessário, deve-se primeiro trazer todos os elementos para uma variável através do comando SELECT INTO. Depois de feitas as alterações, a coluna inteira deve ser atualizada. Este procedimento é mostrado no bloco anônimo (linhas 28 a 41). Após a execução do bloco anônimo, o resultado é:

```
551122457864
5511972345677
5511997662134
```



## TRANSAÇÕES E PL/SQL

Uma transação é um conjunto de comandos SQL que formam uma unidade indivisível. Cada comando realiza parte da tarefa e é necessário que todos sejam executados para que a tarefa seja concluída com sucesso. O agrupamento de comandos em uma transação informa ao SGBDR que todos os comandos devem ser executados com sucesso ou o estado do banco de dados não é alterado. É mais ou menos como um "tudo ou nada".

No jargão de banco de dados, uma transação deve obedecer aos critérios ACID (*Atomic*, *Consistent*, *Isolated*, *Durable* ou Atômico, Consistente, Isolado e Persistente).

Blocos de programa PL/SQL podem ser configurados como transações autônomas. Isto significa que, mesmo que tenham sido chamados de dentro de outra transação, os comandos de controle da transação considerarão o estado do banco de dados no momento em que o bloco foi iniciado. Para se definir um bloco PL/SQL como uma transação autônoma, deve-se utilizar a diretiva PRAGMA AUTONOMOUS_TRANSACTION. Os blocos PL/SQL que podem ser configurados como transações autônomas são blocos anônimos (apenas o bloco mais externo), funções e *procedures*, tanto individuais quanto em *packages* e *triggers*.

Transações autônomas devem ser utilizadas em situações onde o bloco contém diversos comandos SQL que alteram o estado do banco de dados e se deseja desfazer estas alterações em determinadas situações (em caso de erro, por exemplo).

Em uma transação, todas as alterações feitas em tabelas de um banco de dados são tornadas persistentes, ou seja, passam a representar o novo estado do banco de dados para todos os usuários quando um COMMIT é executado. Enquanto um COMMIT não é executado, todas as alterações feitas podem ser revertidas através do comando ROLLBACK. O exemplo a seguir ilustra o uso de transações autônomas.

```
CREATE TABLE tab1 (
    a1 NUMBER,
    a2 NUMBER
);

CREATE TABLE tab2 (
    b1 NUMBER,
    b2 NUMBER
);

CREATE OR REPLACE PROCEDURE exemplo_transacao (
    aa1 NUMBER,
    aa2 NUMBER,
    bb1 NUMBER,
    bb2 NUMBER,
    commit_ou_rollback NUMBER
)
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO tab1 VALUES (aa1, bb1);
    INSERT INTO tab2 VALUES (bb1, bb2);
    IF commit_ou_rollback = 1 THEN
        COMMIT;
    ELSE
        ROLLBACK;
    END IF;
END;

BEGIN
    exemplo_transacao(1, 2, 10, 20, 1);
    exemplo_transacao(3, 4, 30, 40, 0);
    exemplo_transacao(5, 6, 50, 60, 1);
END;
```

Com as tabelas *TAB1 e TAB2* inicialmente vazias, a primeira chamada à *procedure* insere um registro em cada e executa um COMMIT. Na

segunda chamada, os registros são inseridos, mas um ROLLBACK é executado, fazendo com que o estado das tabelas retorne ao [illegible], era antes das duas inserções. A terceira chamada insere mais um registro em cada tabela. O conteúdo das duas tabelas, após a execução do bloco anônimo, é mostrado a seguir.

| | A1 | A2 |
|---|---|---|
| 1 | 5 | 50 |
| 2 | 1 | 10 |

TAB1

| | B1 | B2 |
|---|---|---|
| 1 | 50 | 60 |
| 2 | 10 | 20 |

TAB2

## Atividade Extra

O SGBDR Oracle não permite que seja declarado um tipo RECORD no nível de esquema. Existe o tipo OBJECT que pode emular um tipo RECORD. Pesquise os conceitos básicos do tipo OBJECT do ORACLE e tente refazer a questão discursiva fazendo com que a função *consulta_filme* retorne todos os campos das linhas retornadas.

## Referência Bibliográfica

ELMASRI, R. e NAVATHE, S. B. **Sistemas de Banco de Dados**. 7ª Ed., São Paulo: Pearson, 2011.

PUGA, S., FRANÇA, E. e GOYA, M. **Banco de Dados: Implementação em SQL, PL SQL e Oracle 11g**. São Paulo: Pearson, 2014.

GROFF, J. R., WEINBERG, P. N. e OPPEL, A. J. **SQL: The Complete Reference.** 3ª Ed., Nova York: McGraw-Hill, 2009.

FEUERSTEIN, S. **Oracle PL/SQL Programming**. 6ª Ed., O'Reilly, 2014.

Gonçalves, E. **PL/SQL: Domine a linguagem do banco de dados Oracle**. Versão Digital. Casa do Código, 2015.

[1] É possível utilizar comandos DDL em PL/SQL utilizando-se SQL dinâmico (EXECUTE IMMEDIATE). No entanto, além de problemas relacionados a segurança e desempenho, alguns autores não recomendam o seu uso.

**Ir para exercício**